腾讯云加社区 RP 18.7k 发布于 腾讯云技术社区 关注专栏
2019-01-02 发布

MySQL 索引及查询优化总结

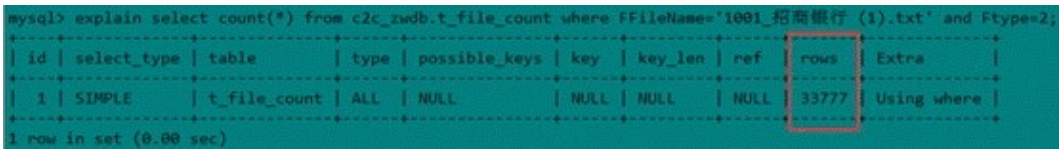
原创 程序员 sql mysql优化 mysql 733 次阅读 · 读完需要 26 分钟

> 本文由云+社区发表

文章《MySQL查询分析》讲述了使用MySQL慢查询和explain命令来定位mysql性能瓶颈的方法，定位出性能瓶颈的sql语句后，则需要对低效的sql语句进行优化。本文主要讨论MySQL索引原理及常用的sql查询优化。

一个简单的对比测试

前面的案例中，c2c_zwdb.t_file_count表只有一个自增id，FFileName字段未加索引的sql执行情况如下：

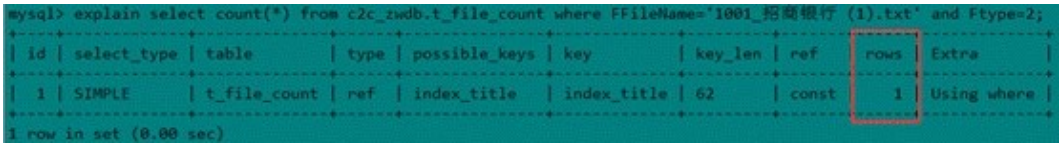


在上图中，type=all，key=null，rows=33777。该sql未使用索引，是一个效率非常低的全表扫描。如果加上联合查询和其他一些约束条件，数据库会疯狂的消耗内存，并且会影响前端程序的执行。

这时给FFileName字段添加一个索引：

alter table c2c_zwdb.t_file_count add index index_title(FFileName);

再次执行上述查询语句，其对比很明显：



在该图中，type=ref，key=索引名（index_title），rows=1。该sql使用了索引index_title，且是一个常数扫描，根据索引只扫描了一行。

比起未加索引的情况，加了索引后，查询效率对比非常明显。

MySQL索引

通过上面的对比测试可以看出，索引是快速搜索的关键。MySQL索引的建立对于MySQL的高效运行是很重要的。对于少量的数据，没有合适的索引影响不是很大，但是，当随着数据量的增加，性能会急剧下降。如果对多列进行索引(组合索引)，列的顺序非常重要，MySQL仅能对索引最左边的前缀进行有效的查找。

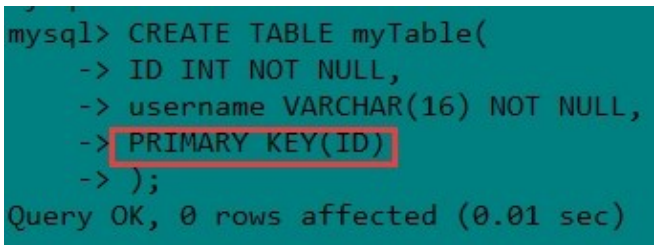
下面介绍几种常见的MySQL索引类型。

索引分单列索引和组合索引。单列索引，即一个索引只包含单个列，一个表可以有多个单列索引，但这不是组合索引。组合索引，即一个索引包含多个列。

1、MySQL索引类型

(1) 主键索引 PRIMARY KEY

它是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引。



当然也可以用 ALTER 命令。记住：一个表只能有一个主键。

(2) 唯一索引 UNIQUE

唯一索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。可以在创建表的时候指

```
ALTER TABLE table_name ADD UNIQUE ( column )
```

(3) 普通索引 INDEX

这是最基本的索引，它没有任何限制。可以在创建表的时候指定，也可以修改表结构，如：

```
ALTER TABLE table_name ADD INDEX index_name ( column )
```

(4) 组合索引 INDEX

组合索引，即一个索引包含多个列。可以在创建表的时候指定，也可以修改表结构，如：

```
ALTER TABLE table_name ADD INDEX index_name( column1 , column2 , column3 )
```

(5) 全文索引 FULLTEXT

全文索引（也称全文检索）是目前搜索引擎使用的一种关键技术。它能够利用分词技术等多种算法智能分析出文本文字中关键字词的频率及重要性，然后按照一定的算法规则智能地筛选出我们想要的搜索结果。

可以在创建表的时候指定，也可以修改表结构，如：

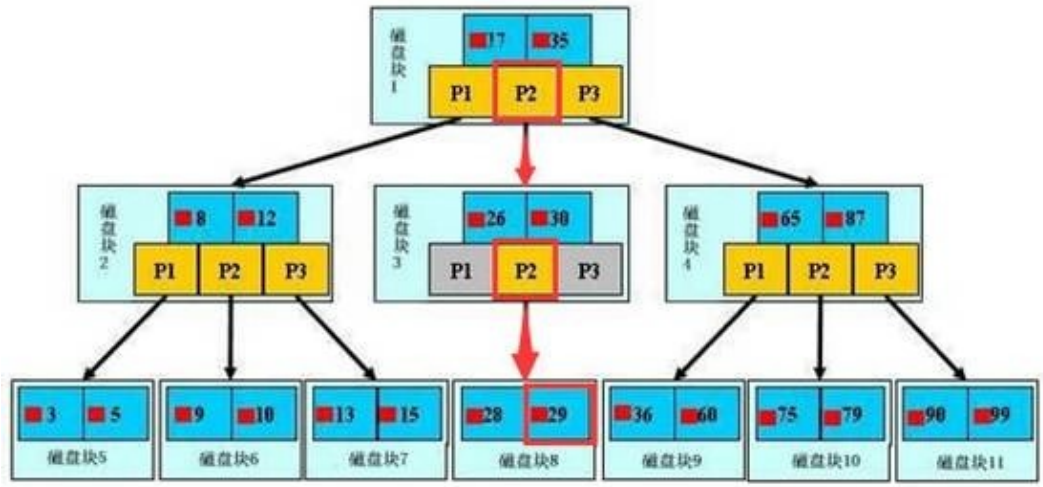
```
ALTER TABLE table_name ADD FULLTEXT ( column )
```

2、索引结构及原理

mysql中普遍使用B+Tree做索引，但在实现上又根据聚簇索引和非聚簇索引而不同，本文暂不讨论这点。

b+树介绍

下面这张b+树的图片在很多地方可以看到，之所以在这里也选取这张，是因为觉得这张图片可以很好的诠释索引的查找过程。



如上图，是一颗b+树。浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。

真实的数据存在于叶子节点，即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

查找过程

在上图中，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

性质

(1) 索引字段要尽量的小。

通过上面b+树的查找过程，或者通过真实的数据存在于叶子节点这个事实可知，IO次数取决于b+数的高度h。

假设当前数据表的数据量为N，每个磁盘块的数据项的数量是m，则树高h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；

而m = 磁盘块的大小/数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的；如果数据项占的空间越小，数据项的数量m越多，树的高度h越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。

(2) 索引的最左匹配特性。

当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

建索引的几大原则

(1) 最左前缀匹配原则

对于多列索引，总是从索引的最前面字段开始，接着往后，中间不能跳过。比如创建了多列索引(name,age,sex)，会先匹配name字段，再匹配age字段，再匹配sex字段的，中间不能跳过。mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配。

一般，在创建多列索引时，where子句中使用最频繁的一列放在最左边。

看一个补符合最左前缀匹配原则和符合该原则的对比例子。

实例：表c2c_db.t_credit_detail建有索引(`Flistid` , `Fbank_listid`)

```
mysql> show create table t_credit_detail\G
*************************** 1. row ***************************
       Table: t_credit_detail
Create Table: CREATE TABLE `t_credit_detail` (
  `Flistid` varchar(32) NOT NULL DEFAULT '',
  `Fcoding` varchar(32) NOT NULL DEFAULT '',
  `Fspid` varchar(16) NOT NULL DEFAULT '',
  `Fbank_listid` varchar(32) NOT NULL DEFAULT '',
  `Fbank_backid` varchar(32) NOT NULL DEFAULT '',
  `Fbuy_uid` bigint(20) DEFAULT NULL,
  `Fbuyid` varchar(65) NOT NULL DEFAULT '',
  `Fbuy_name` varchar(64) NOT NULL DEFAULT '',
  `Fbuy_bank_type` smallint(6) NOT NULL DEFAULT '0',
  `Fsale_uid` bigint(20) DEFAULT NULL,
  `Fsaleid` varchar(65) NOT NULL DEFAULT '',
  `Fsale_name` varchar(64) DEFAULT NULL,
  `Fpaynum` bigint(20) DEFAULT NULL,
  `Fcreate_time` datetime DEFAULT NULL,
  `Fpay_time` datetime DEFAULT NULL,
  `Fip` varchar(16) DEFAULT NULL,
  `Fmemo` varchar(128) DEFAULT NULL,
  `Fexplain` varchar(128) DEFAULT NULL,
  `Fmodify_time` datetime DEFAULT NULL,
  `Ftrade_type` smallint(6) DEFAULT NULL,
  `Fbank_pay_time` datetime DEFAULT NULL,
  `Fextend` text,
  PRIMARY KEY (`Flistid`,`Fbank_listid`),
  KEY `idx_buy_uid_bank_pay_time` (`Fbuy_uid`,`Fbank_pay_time`),
  KEY `idx_bank_pay_time` (`Fbank_pay_time`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

不符合最左前缀匹配原则的sql语句：

select * from t_credit_detail where Fbank_listid='201108010000199'G

该sql直接用了第二个索引字段Fbank_listid，跳过了第一个索引字段Flistid，不符合最左前缀匹配原则。用explain命令查看sql语句的执行计划，如下图：

```
mysql> explain select * from t_credit_detail where Fbank_listid='201108010000199'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t_credit_detail
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 131
        Extra: Using where
1 row in set (0.00 sec)
```

从上图可以看出，该sql未使用索引，是一个低效的全表扫描。

符合最左前缀匹配原则的sql语句：

select * from t_credit_detail where Flistid='2000000608201108010831508721' and Fbank_listid='201108010000199'G

缀匹配原则。用explain命令查看sql语句的执行计划，如下图：

从上图可以看出，该sql使用了索引，仅扫描了一行。

对比可知，符合最左前缀匹配原则的sql语句比不符合该原则的sql语句效率有极大提高，从全表扫描上升到了常数扫描。

(2) 尽量选择区分度高的列作为索引。

比如，我们会选择学号做索引，而不会选择性别来做索引。

(3) =和in可以乱序

比如a = 1 and b = 2 and c = 3，建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式。

(4) 索引列不能参与计算，保持列“干净”

比如：Flistid+1> ‘20000006082011080108310508721 ‘。原因很简单，假如索引列参与计算的话，那每次检索时，都会先将索引计算一次，再做比较，显然成本太大。

(5) 尽量的扩展索引，不要新建索引。

比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

索引的不足

虽然索引可以提高查询效率，但索引也有自己的不足之处。

索引的额外开销：

(1) 空间：索引需要占用空间；

(2) 时间：查询索引需要时间；

(3) 维护：索引须要维护（数据变更时）；

不建议使用索引的情况：

(1) 数据量很小的表

(2) 空间紧张

常用优化总结

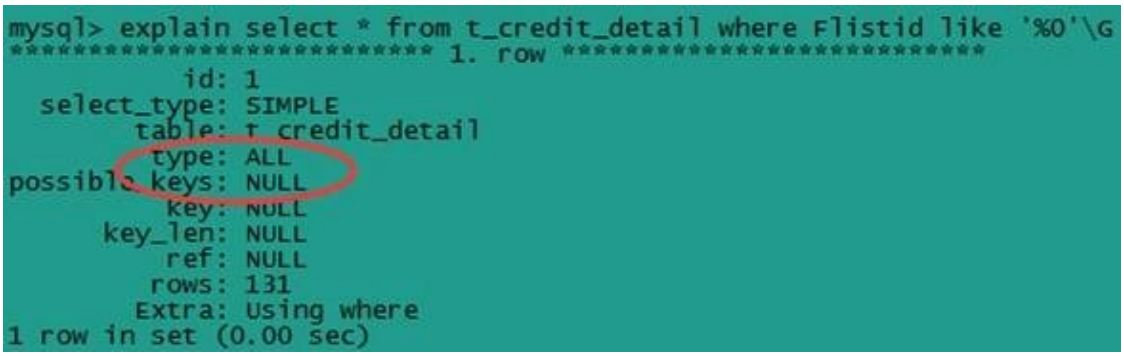
优化语句很多，需要注意的也很多，针对平时的情况总结一下几点：

1、有索引但未被用到的情况（不建议）

(1) Like的参数以通配符开头时

尽量避免Like的参数以通配符开头，否则数据库引擎会放弃使用索引而进行全表扫描。

以通配符开头的sql语句，例如：select * from t_credit_detail where Flistid like '%0'G



这是全表扫描，没有使用到索引，不建议使用。

不以通配符开头的sql语句，例如：select * from t_credit_detail where Flistid like '2%'G

```
mysql> explain select * from t_credit_detail where Flistid like '2%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t_credit_detail
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 34
          ref: NULL
         rows: 12
        Extra: Using where
1 row in set (0.00 sec)
```

很明显，这使用到了索引，是有范围的查找了，比以通配符开头的sql语句效率提高不少。

(2) where条件不符合最左前缀原则时

例子已在最左前缀匹配原则的内容中有举例。

(3) 使用！= 或 <> 操作符时

尽量避免使用！= 或 <>操作符，否则数据库引擎会放弃使用索引而进行全表扫描。使用>或<会比较高效。

select * from t_credit_detail where Flistid != '20000006082011080108310831508721'G

```
mysql> explain select * from t_credit_detail where Flistid != '20000006082011080108310831508721'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t_credit_detail
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 34
          ref: NULL
         rows: 131
        Extra: Using where
1 row in set (0.00 sec)
```

(4) 索引列参与计算

应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。

select * from t_credit_detail where Flistid +1 > '20000006082011080108310831508722'G

```
mysql> explain select * from t_credit_detail where Flistid +1 > '2000000608201108010831508722'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t_credit_detail
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 131
        Extra: Using where
1 row in set (0.00 sec)
```

(5) 对字段进行null值判断

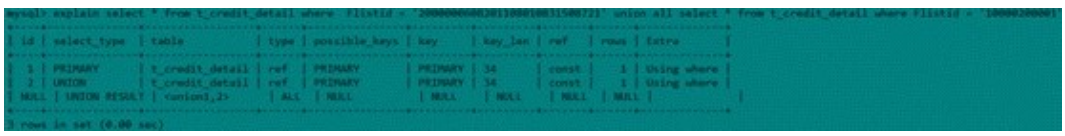
应尽量避免在where子句中对字段进行null值判断，否则将导致引擎放弃使用索引而进行全表扫描，如： 低效：select * from t_credit_detail where Flistid is null ;

可以在Flistid上设置默认值0，确保表中Flistid列没有null值，然后这样查询： 高效：select * from t_credit_detail where Flistid =0;

(6) 使用or来连接条件

应尽量避免在where子句中使用or来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如： 低效：select * from t_credit_detail where Flistid = '2000000608201108010831508721' or Flistid = '10000200001';

可以用下面这样的查询代替上面的 or 查询： 高效：select *from t_credit_detail where Flistid = '2000000608201108010831508721' union all select* from t_credit_detail where Flistid = '10000200001';



2、避免select *

在解析的过程中，会将'*' 依次转换成所有的列名，这个工作是通过查询数据字典完成的，这意味着将耗费更多的时间。

所以，应该养成一个需要什么就取什么的好习惯。

3、order by 语句优化

方法：1.重写order by语句以使用索引；

2.为所使用的列建立另外一个索引

3.绝对避免在order by子句中使用表达式。

4、GROUP BY语句优化

提高GROUP BY 语句的效率, 可以通过将不需要的记录在GROUP BY 之前过滤掉

低效:

SELECT JOB , AVG(SAL)

FROM EMP

GROUP by JOB

HAVING JOB = ‘PRESIDENT'

OR JOB = ‘MANAGER'

高效:

SELECT JOB , AVG(SAL)

FROM EMP

WHERE JOB = ‘PRESIDENT'

OR JOB = ‘MANAGER'

GROUP by JOB

5、用 exists 代替 in

很多时候用 exists 代替 in 是一个好的选择： select num from a where num in(select num from b) 用下面的语句替换： select num from a where exists(select 1 from b where num=a.num)

6、使用 varchar/nvarchar 代替 char/nchar

尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

7、能用DISTINCT的就不用GROUP BY

SELECT OrderID FROM Details WHERE UnitPrice > 10 GROUP BY OrderID

可改为:

SELECT DISTINCT OrderID FROM Details WHERE UnitPrice > 10

8、能用UNION ALL就不要用UNION

UNION ALL不执行SELECT DISTINCT函数，这样就会减少很多不必要的资源。

9、在Join表的时候使用相当类型的例，并将其索引

如果应用程序有很多JOIN 查询，你应该确认两个表中Join的字段是被建过索引的。这样，MySQL内部会启动为你优化Join的SQL语句的机制。

而且，这些被用来Join的字段，应该是相同的类型的。例如：如果你要把 DECIMAL 字段和一个 INT 字段Join在一起，MySQL就无法使用它们的索引。对于那些STRING类型，还需要有相同的字符集才行。（两个表的字符集有可能不一样）

你可能感兴趣的

- **mysql查询与索引优化优化1** 演绎梦幻舞步 mysql
- **干货满满，腾讯云+社区技术沙龙 Kafka Meetup 深圳站圆满结束** 腾讯云加社区 github
- **Mysql索引优化** waterandair mysql
- **Mysql 架构及优化之-索引优化** Julylovin mysql
- **MySQL索引原理及慢查询优化** mysql mysql优化 mysql索引
- **思考线上技术社区** 题叶 社区
- **MySQL5.6子查询优化总结** bsknight mysql

1 条评论

默认排序 时间排序

北漂青年 · 1月3日

讲的很清楚，明白。谢谢作者

赞 回复

文明社会，理性评论

发布评论